PhatKV: Towards an Efficient Metadata Engine for KV-based File Systems on Modern SSD

Yiwen Zhang, Guokuan Li*, Kai Lu*, and Jiguang Wan
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, China
{zhangyiwen, liguokuan, emperorlu, jgwan}@hust.edu.cn

Ting Yao, Huatao Wu, and Daohui Wang
Cloud Storage Service Product Dept
Huawei Technologies Co.,Ltd
Shenzhen, China
{yaoting17, wuhuatao, wangdaohui}@huawei.com

***Abstract*—Emerging file systems are built with KV stores as the storage engine of metadata, leading to KV workloads with unique characteristics such as a significant proportion of single-key lookups, simpler scan operations, and locality on key access. LSM-Tree based KV stores are widely used to store file system metadata. However, these KV stores designed for general purposes are unable to reach their full potential when serving as the storage engine of file system metadata and face challenges such as 1) limited write performance because of excessive disk IOs and CPU resource consumption caused by key sorting and outdated IO interface; 2) inefficient key searching caused by unsuitable index structures. We propose PhatKV, a KV store specifically tailored to function as file system metadata storage on modern SSD. PhatKV proposes a new Piece Hash Index to facilitate retrieving KV items via a two-level hash structure with limited memory consumption by tactfully moving parts of the index structures between SSD and memory. PhatKV also eliminates the sorting overhead, such as disk IOs and CPU resource utilization, by aggregating KV items of each directory on SSD via a Three-stage Aggregation strategy, and enhances the data IO performance by leveraging the new io_uring interface. Experimental results show that throughput of PhatKV outperforms state-of-the-art KV stores by 1.2× to 7.9× on KV operations. File systems using PhatKV achieve improvement on throughput of metadata operations by 1.6× to 3.1×.**

***Index Terms*—key-value store, file system, metadata, SSD**

I. Introduction

File systems play a crucial role in managing data for various applications. The metadata of a file system contains data of the files' attributes and the file system hierarchy. File system metadata is vital for ensuring the overall performance of file systems since metadata operations account for the majority among all the file system operations [1]. Recent works have proposed leveraging key-value (KV) stores as the storage engine for file system metadata and built KV-based file systems [1]–[6], due to the high performance, efficient interface, and low consistency overhead of KV stores. With this method, the directory hierarchy and file system metadata are transformed into KV entries (i.e., $\langle pinode\_fname, stat \rangle$) [1]–[3], [5], [6]. The KV operations performed on the file system metadata present three unique characteristics: 1) Most KV operations are single-key operations, and the proportion of query is up to 90% in some workloads; 2) Range scans only need to return unsorted KV entries with the same pinode; 3) KV items are accessed with directory locality, meaning that keys with the same pinode are more likely to be accessed at adjacent times.

*Corresponding author

LSM-Tree based KV stores such as LevelDB and RocksDB are popular KV stores on SSD. These KV stores support comprehensive types of KV operations and fast write processing. Thus, LSM-Tree based KV stores are widely used as the storage engine of metadata when building KV-based file systems. However, existing LSM-Tree KV stores and their optimized variants are mainly designed for general use without considering the workload characteristics of metadata operations. There is still room for improvement in the efficiency of KV operations and metadata operations by tailoring the KV stores according to the file system metadata workloads in both write and read.

LSM-Tree KV stores are not the best option for file systems due to three restrictions when it comes to the access patterns of file system metadata. First, they can't achieve higher write speed because of the sorting overhead of LSM-Tree. LSM-Tree KV stores regularly read and write data at each level to sort KV data on SSD. Both IO amplification and merging KV data are incurred in the background compaction processes [7], [8] causing excessive SSD IOs and CPU overhead. The background compaction slows down the foreground processing of KV requests. Moreover, existing LSM-Trees use outdated POSIX-style synchronous IO interface, which further downgrades the efficiency of IO and background data processing. Second, the multi-level structure of LSM-Tree reduces the effectiveness of key searching. Locating a key in LSM-Tree can be complicated because of the multi-level organization of SSTable and the complex index block parsing [9]. Additionally, LSM-Tree is not directory-aware in two aspects: 1) The latency of accessing KV entries from a directory varies greatly because KV entries from a directory may go to different SSTables or even different levels. 2) All the metadata of SSTables must be kept in memory to guarantee efficient key searching, which results in memory overhead. Third, although the file systems do not require the sorted results, LSM-Tree incurs overhead for sorting the KV entries and returns a sorted sequence of KV entries for a scan operation.

To address the limitations of the aforementioned KV stores, we propose PhatKV, a tailored KV store designed especially for building KV-based file systems with efficient metadata management. PhatKV achieves this through three essential designs. First, we design the **Piece Hash Index** to efficiently index keys with restricted memory consumption. It is a two-level hash index structure and allows efficient key retrieval through two hash lookups, while selectively maintaining necessary parts of the piece hash index in memory by leveraging the directory locality. Second, we present a **Three-stage Aggregation** strategy for managing data on disk to minimize the disk IOs and CPU resource consumption while still supporting the scan operations by aggregating KV items rather than sorting them. Third, PhatKV leverages the new io_uring [10] interface to enhance disk IO performance and fully unleash the capabilities of modern SSD.

The primary contributions of PhatKV are as follows:

- **Piece Hash Index.** We propose the new piece hash index for the file system metadata. It enables more efficient retrieval of KV items and lower memory footprint.
- **Three-stage Aggregation.** We present the three-stage aggregation strategy to effectively manage file system metadata on modern SSD. It aggregates KV items from each directory instead of sorting all the KV items, which reduces the SSD IO required to manage KV items on the SSD.
- **Asynchronous IO via io_uring.** We implement the aggregation via asynchronous io_uring interface. It further enhances the efficiency of aggregation.
- We implement and evaluate PhatKV. The results demonstrate that PhatKV's throughput outperforms LevelDB and RocksDB by 1.2× to 7.9×, respectively. Additionally, it also achieves 1.2× to 1.5× higher throughput and four times less memory footprint compared to KVell. Moreover, file system using PhatKV achieves 1.6× to 3.1× improvements on metadata operations' throughput.

II. Background and Motivation

A. *KV-based file systems*

File systems are widely used to manage data in the form of files and directories within a directory tree across various scenarios. File systems store the metadata of files (directories are considered as special files) and the directory tree structure to facilitate file data retrieval. To achieve fast file access, efficient access to file system metadata is critical. Key-value (KV) stores are deployed as the storage engine of emerging large-scale file systems for managing file system metadata because of exceptional performance and scalability [1]–[6], [11]. Integrating KV stores and building KV-based file systems provides scalable and more efficient metadata services.

File system metadata operations are converted into KV operations when constructing a KV-based file system. The file system metadata, i.e., file metadata and directory tree structure, is then converted into KV pairs in the KV stores. The file system information is commonly represented by $\langle pinode\_fname \rangle$, in which the term "pinode" denotes the inode number of the

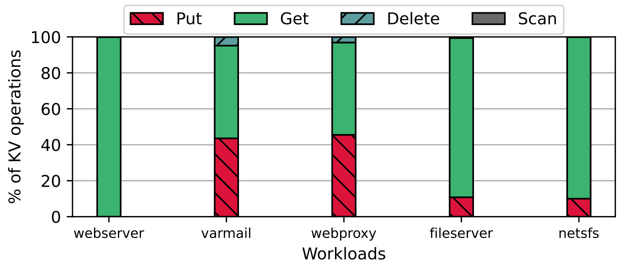

Fig. 1. **Proportion of different KV operations.** This figure illustrates the distribution of various KV operations across different file system workloads.

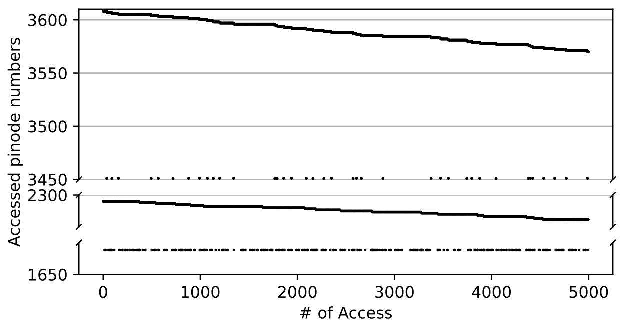

Fig. 2. **Statistics of pinode access from WebProxy workload.** The x-axis represents each key access from file system operations. The y-axis displays the corresponding pinode of the accessed keys.

parent directory of a particular file or directory [1]–[3], [5], [6]. Accessing a file with a given path is implemented by recursive parsing of directories along the path. For example, locating a file at "/home/user/data" can be transferred into three get operations for the KV stores: Get(0_home), Get(1_user), and Get(2_data). The special format of KV pairs and the usage scenario lead to KV workloads with distinctive characteristics for KV stores under a file system:

First, single-key operations account for the largest proportion of all KV operation calls, particularly the get operations. As shown in Figure 1, the get operations constitute up to 90% of all KV operations [12]. This is primarily due to the fact that accessing a file requires recursively parsing the file's path by conducting key searches within the KV stores. Moreover, creating a new file leads to the insertion of a new KV item to the KV store.

Second, when accessing KV entries, single-key operations from the file system exhibit locality, which we refer to as "directory locality". Figure 2 illustrates the pinode of accessed key from a synthetic workload from filebench. We can observe that: (1) some pinodes of KV entries are always accessed during the test as they are the root of the sub-directories; (2) some pinodes of accessed KV entries remain unchanged for a short period before shifting to another pinode in the subsequent period. The locality arises because file system workloads typically operate within specific working directories. Thus, KV entries associated with the same pinode are more likely to be accessed in the KV store over a period of time, resulting in

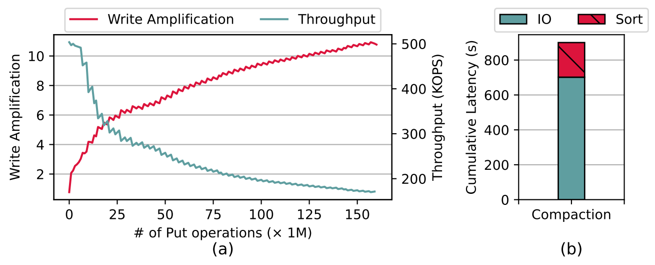

Fig. 3. **Analysis of write operations.** (a) shows the variation of foreground throughput and background write amplification under each 1 million insert operations. (b) presents the latency breakdown of compaction for KV merge and data IO.

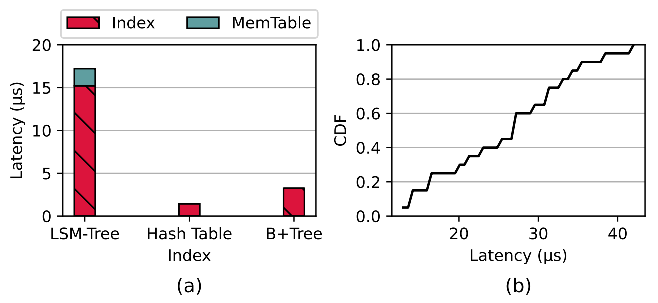

Fig. 4. **Analysis of read operations.** (a) shows the latency of locating a key in LevelDB, in-memory hash table, and in-memory B+Tree. (b) shows the CDF of the latency for accessing KV entries from a single directory.

locality on a range of KV entries.

Third, the file system does not require the KV store to sort all KV pairs. Since KV entries in different directories are isolated, sorting the KV entries from different directories is worthless. Furthermore, the scan operation required by the file system differs from a regular scan. The file system retrieves all KV entries associated with a specified pinode, without regard to their sequence. In contrast, KV stores typically yield a sorted sequence of KV pairs within a user-defined key range. KV stores for file systems can sacrifice the total order of KV entries for a better performance.

B. LSM-Tree

LSM-Tree [13] based KV stores, such as LevelDB[1] and RocksDB[2], are designed for fast SSD and are widely used as the backbones of various storage systems [14]. Because LSM KV stores support extensive KV operations including Put, Get, and Scan, and offer superior write performance on SSD, developers building KV-based file systems choose to use them as the metadata storage engine. The basic structure of LSM-Tree KV stores includes an in-memory component called Memtable, and multiple on-disk components called SSTables. The SSTables are organized in a multi-level structure. In each level except for level 0, the key range of each SSTable is not overlapped. The total size of level $N+1$ (L_{N+1}) is r times larger than that of level N (L_N), where r is typically set to 10. New KV items are inserted to the Memtable first and then moved to the higher level at the granularity of the SSTable by the background compaction process. Locating a key requires searching level by level, starting from the Memtable, until the target key is found or the last level on SSD is reached.

C. Motivation

LSM KV stores are designed for general use cases where strict sorting of KV entries is indispensable. However, when combined with the specific scenario for file system metadata storage, the design choice of LSM-Tree is not always optimal and limits the write and read performance, respectively.

Overhead of sorting limits the write performance. LSM-Tree must sort the KV entries on the SSD to guarantee the strict order of keys at each level. The KV entries are sorted and merged by the background compaction threads. Periodically, compaction threads select an SSTable at L_N and one or more SSTables at L_{N+1} that have the overlapped key range with the SSTable at L_N, and read them into memory, merge them to construct new SSTables, and finally write the new SSTables to L_{N+1}. The compaction process incurs a huge number of data writes that are significantly larger than the user's writes (i.e., write amplification), reducing the efficiency of foreground operations. As shown in Figure 3(a), we evaluate LevelDB by loading 160 million KV entries and recording the write throughput and write amplification every 1 million put operations. The write amplification of LSM-Tree increases with the size of the inserted data, leading to a downgraded write performance. The CPU overhead for sorting KV entries in memory is also incurred during the compaction operation. We record the time spent on the KV entries merging during the compaction process as shown in Figure 3(b). The merge process accounts for 22% of the total compaction time. Additionally, existing LSM-Tree KV stores typically access the underlying storage device via a POSIX synchronous interface. Because of the overhead of an inefficient IO stack, redundant data copy, and time wasted waiting for IO requests to complete, KV stores with the outdated IO interface are unable to effectively exploit modern fast storage devices.

Multi-level tree structure is not optimal for key searching and scan. The structure of LSM-Tree is not optimal for accessing KV entries of file system metadata. First, the multi-level organization of LSM-Tree increases the search latency, while the underlying structure of SSTables further slows down key searching. Figure 4(a) shows the average time spent on locating SSTable files and locating keys in SSTables in LevelDB. We compare the latency of key locating in LevelDB with that of an in-memory hash table and that of an in-memory B+Tree. The time for locating a key in LevelDB is 9.7× and 4.3× longer than that of the hash table and B+Tree, respectively. Second, the LSM-Tree is not directory-aware, which means KV entries from a directory may scatter across different SSTables or levels, leading to varied access latency when accessing KV entries in a directory. We collect the latency of accessing KV entries from the same directory

[1]https://github.com/google/leveldb
[2]https://github.com/facebook/rocksdb

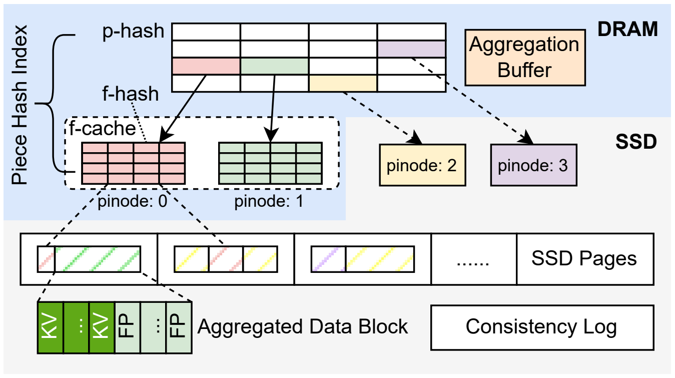

Fig. 5. **Overall structure of PhatKV.** In this figure, the piece hash index contains 4 f-hash structures. The f-hash of pinode 0 and 1 are in the f-cache, while those of pinode 2 and 3 are stored on SSD. Pinode 0 has two data blocks on SSD.

via LevelDB. The size of a directory is set to 20, which means there are 20 KV entries for each pinode. We randomly choose a directory and read all the KV entries belong to that directory (i.e., have the same pinode). Figure 4(b) shows the cumulative distribution function (CDF) of the latency. The large variation in latency indicates that the LSM-Tree is unaware of the access from a directory. Additionally, to guarantee the efficiency of key searching, all the index blocks of SSTables must be kept in memory, which also increases the memory overhead. Third, to gather KV entries from a directory, the scan operations of LSM-Tree based KV stores must merge and sort related SSTables and data blocks to obtain a sorted KV sequence, which adds more sorting overhead.

III. Design

In this section, we present PhatKV, a KV store designed to work as the metadata storage engine for building KV-based file systems that provide efficient metadata services on modern SSDs. PhatKV aims to address the challenges identified in Section II-C: 1) the sorting overhead from LSM-Tree compaction; 2) the inefficient key searching of LSM-Tree structure. The design of PhatKV follows three principles:

- leveraging a hash-based index and the directory locality to ensure fast queries of frequently accessed KV items;
- aggregating KV entries with the same pinode instead of sorting all the KV items;
- utilizing modern asynchronous storage API to unleash the performance of modern SSDs.

Figure 5 depicts the overall structure of PhatKV. Each file system metadata corresponds to a KV item in PhatKV. The main components of PhatKV include 1) Piece Hash Index: a two-level hash index that stores the KV index entries. Each KV index entry consists of the key hash and value address of the corresponding KV item on the SSD; 2) Aggregated Data Blocks: data blocks that store KV items on the SSD with the same pinode; 3) Aggregation Buffer: an in-memory buffer that proactively aggregates KV items with the same pinode to facilitate further aggregation; 4) Consistency Log: a log that records modifications made to in-memory structures to ensure their consistency.

A. Directory-aware Piece Hash Index

While typical in-memory hash tables are more efficient than both LSM-Tree and B+Tree, storing all of the KV index entries by hash tables makes it difficult to manage scan operations and uses too much memory when working with large datasets. We propose a new structure called the **Piece Hash Index**, which is designed specifically for managing file system metadata.

1) Two-level hash structure: The piece hash index is a directory-aware two-level hash index, as depicted in Figure 5. It consists of a high-level **Pinode Hash (p-hash)** and multiple low-level **File Name Hash (f-hash)** structures. Each f-hash stores the KV index entries with the same pinode. That means that the metadata of files under a directory is handled by the corresponding f-hash structure. The p-hash organizes all the f-hash structures and records the mapping between pinode numbers and the corresponding f-hash structures.

The high-level p-hash is an in-memory hash table. We select the CLHT (Cache-Line Hash Table) [15] as the implementation of the p-hash, because it is a representative and fast in-memory hash table. The p-hash is placed in memory for two reasons: 1) directory operations (such as statdir, mkdir, and rmdir) are not common in real file system workloads [1], and thus the p-hash is less likely to be modified; 2) the number of directories is typically smaller than that of files, so the size of p-hash is relatively small. Our experimental results indicate that the p-hash consumes only 256 MB of memory for 8 million directories.

Each f-hash is associated with a pinode number and stores KV index entries, which consist of an 8-byte hash value for the fname and an 8-byte descriptor for the value address on the SSD (page id and page offset).

2) Directory locality based f-cache: Similar to a typical hash table, storing all the f-hash structures in memory also leads to an excessive memory footprint and recovery overhead. Note that KV entries with the same pinode are accessed at adjacent times, according to the directory locality. Thus, we take advantage of directory locality by just keeping necessary f-hash structures in memory. When a directory is accessed as the current working directory for the first time, all the KV index entries for files in that directory can be loaded into memory by moving the corresponding f-hash into memory. When a directory is no longer accessed, we can remove all the entries by moving the corresponding f-hash from memory to SSD, thus saving memory. As shown in Figure 5, we introduce the f-cache, which uses an LRU list to manage the in-memory f-hash structures. If the memory usage of the f-cache exceeds its pre-defined capacity (i.e., 1 GB in our default configuration), certain f-hash structures will be selected from f-cache and evicted to the underlying SSD.

Because all of the KV index entries are intermixed, a typical hash index cannot have fine-grained control on maintaining the necessary f-hash in memory. However, by combining the p-hash and f-hash structures, the piece hash index achieves this

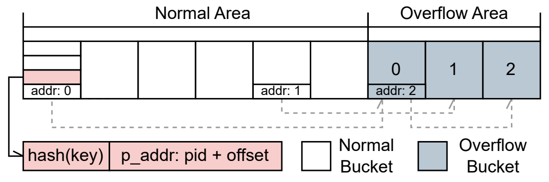

Fig. 6. **Structure of f-hash.** In this figure, the f-hash contains 6 normal buckets and 3 overflow buckets. Bucket 0 links two overflow buckets, and bucket 4 links 1 overflow bucket.

level of fine-grained control. The piece hash index cooperates with the f-cache to achieve efficient key searching with minimal memory overhead.

3) Preventing serialization overhead: The detailed structure of the f-hash is illustrated in Figure 6. Each f-hash consists of multiple buckets, each of which has a size equal to that of a cacheline (i.e., 64 bytes). A bucket contains 4 KV index entries. The incorporation of f-cache may lead to two possible locations for an f-hash, thereby resulting in overhead for compacting the f-hash to an SSD page (serialization) or rebuilding it from data stored in SSD pages (de-serialization). These serialization overheads increase the latency of managing f-hash in the f-cache, i.e., loading and eviction. To eliminate the serialization overhead, we propose three f-hash design principles. First, the f-hash is constructed on a continuous memory space, which is divided into a normal area and an overflow area. The normal area comprises buckets that are accessed by the hash value of the keys. Second, a new overflow bucket that is allocated sequentially from the overflow area is catenated to handle the overflow of f-hash buckets. Third, the f-hash utilizes offsets rather than memory addresses to indicate an overflow bucket. By employing these three designs, the f-hash can be seamlessly moved between SSD and memory without incurring serialization overhead.

4) Adaption to directory with different sizes: Since the SSD's default access unit is 4 KB, any reading or writing of an f-hash smaller than 4 KB will waste SSD IO. To avoid this, we suggest an adaptive combination for small f-hash. We define that the size of f-hash is aligned to 64 bytes, and their sizes double each time they expand. When an f-hash smaller than 4 KB is selected to be evicted from the f-cache, we will combine several f-hash structures into a 4 KB block. Two principles guide the choice of the f-hash for the combination: 1) they are selected to be evicted; 2) the f-hash structures have adjacent pinode numbers. Since nearby pinode numbers are more likely to be accessed, the f-hash structures with those pinode numbers are combined first. When we load an f-hash into memory, we can save SSD IO by using the adaptive combination to pre-fetch the f-hash structures in the same SSD page. On the other hand, accessing entries in a large f-hash may waste IO because the entire f-hash would need to be read into memory. To save undesired reading, we suggest partial reading. An f-hash larger than 4 KB will be divided into multiple 4 KB pages. The target key can be first mapped to a 4 KB page by its hash value. Then, rather than reading the whole f-hash, the partial read strategy allows only loading that page into memory.

B. *Light-weight Three-stage Aggregation*

For file system metadata, keeping KV items completely sorted on SSD (like LSM-Tree) is not the ideal option. However, the piece hash index alone cannot support the scan operations effectively. Since scan operations from file systems do not require the sorting of the result, PhatKV proposes grouping KV items based on the pinode whenever feasible. This approach reduces the overhead of compaction and improves the efficiency of scan operations.

The fundamental unit for managing KV items in PhatKV is the aggregated data block, which stores KV items with the same pinode. The aggregated data block comprises KV data and fingerprint (FP) data as shown in Figure 5. The FP data contains 2-byte fingerprint values for each KV item within the data block. The maximum size of a data block is limited to 4 KB, equivalent to the size of an SSD page. Smaller blocks can be combined to fill an SSD page. To support scan operations, each pinode has a data block set, an in-memory structure that records a list of data block addresses associated with that pinode. Next, We propose a three-stage aggregation strategy for PhatKV, including memory aggregation, disk aggregation, and deep aggregation, to efficiently manage the KV items on SSD.

Memory Aggregation. In this stage, we utilize an in-memory aggregation buffer to perform a preliminary aggregation for KV items according to their pinode numbers, as shown in Figure 7(a). For each pinode, the aggregation buffer maintains a skiplist, referred to as a pinode set. All pinode sets are sorted in descending order of their sizes. Once the total size of KV items in the aggregation buffer (64MB) or the size of a pinode set exceeds the threshold (4KB), the background aggregation threads are triggered. Aggregation threads select the pinode set with the largest size from the aggregation buffer, construct data blocks, and write these data blocks to the SSD. Whenever a new data block is persisted, the associated KV index entries for these KV items are inserted to the corresponding f-hash. The memory aggregation reserves the pinode with fewer entries in memory for aggregation in the future. This reduces the number of data blocks of a pinode, and thus improves the scan performance.

Disk Aggregation. A data block is added to a pinode set each time after the memory aggregation, and scattered data blocks of a pinode downgrade the efficiency of the scan operation. The disk aggregation stage aims to aggregate data on the SSD to reduce the number of SSD pages that need to be read during a scan operation. Data blocks smaller than 2 KB are regarded as scattered blocks. The background aggregation thread is triggered to collect data from various SSD pages when the number of scattered blocks exceeds a specific threshold, as shown in Figure 7(b). During disk aggregation, all scattered blocks are read and combined into larger ones. While the compaction merges and sorts all KV items from various data blocks, the disk aggregation simply

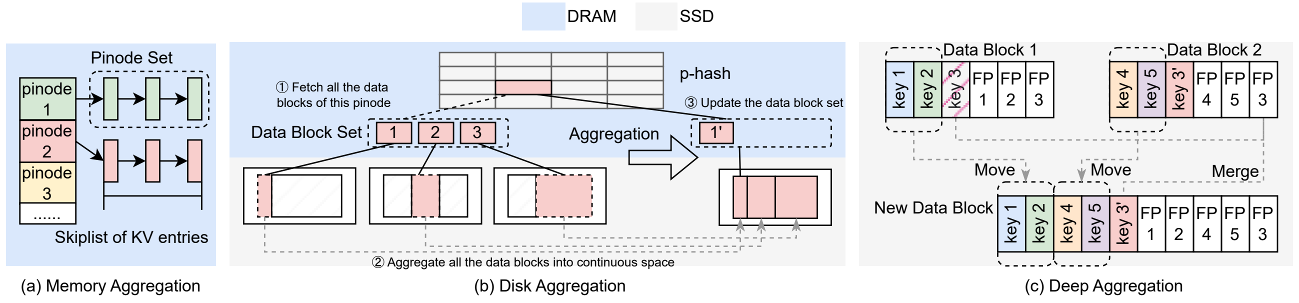

Fig. 7. **The process of three-stage aggregation.** (a) shows the memory aggregation. KV entries are grouped according to pinode by skiplists in memory. (b) demonstrates the process of disk aggregation that combines three data blocks of pinode 3 into a new data block 1'). (c) shows that deep aggregation merges the two data blocks and removes the obsoleted key 3.

gathers scattered blocks from different pages, consolidates them into continuous pages, and then writes those pages. This approach effectively reduces the CPU overhead brought by merging and sorting. Upon flushing the newly aggregated data block, previous data blocks are marked as invalid and subsequently reclaimed.

Deep Aggregation. Since the aggregation does not merge and sort KV entries within a data block, it leaves obsolete data behind, wasting storage space. The deep aggregation stage is subsequently triggered to eliminate invalid data from those data blocks. To keep track of each pinode, we record the ratio of invalid data of each pinode. When this ratio exceeds the threshold, deep aggregation is activated for all data blocks associated with that pinode. PhatKV leverages an FP-based merge operation in deep aggregation to prevent sorting the KV entries in the data blocks as shown in Figure 7(c). The deep aggregation follows four steps. First, it reads all the FP data from all the data blocks and identifies KV items with matching fingerprints. Second, the KV entries with the same FP values are extracted from the data blocks and compared with each other. The older KV entries that are updated by a newer version or deleted entries are removed. Third, valid KV items without fingerprint conflicts can be moved in batch to a new data block, which reduces the CPU overhead of recalculating fingerprints. Fourth, after eliminating the invalid data, the new data block is written back to new SSD pages.

By leveraging the memory aggregation, PhatKV reduces the write amplification for collecting KV entries with the same pinode on SSD. With the disk aggregation and deep aggregation, PhatKV can further cluster KV data of each pinode and remove invalid data with lower CPU overhead.

C. Consistency

To ensure data consistency of in-memory structures, such as p-hash, f-hash, and the aggregation buffer, PhatKV maintains a consistency log on the SSD to record modifications of these structures. The consistency log records the newly inserted KV items (resembles the WAL in an LSM-Tree) and the modifications on the in-memory structures. KV data is considered as consistent once it has been persisted to the consistency log, and can be recovered when the system restarts. The consistency of

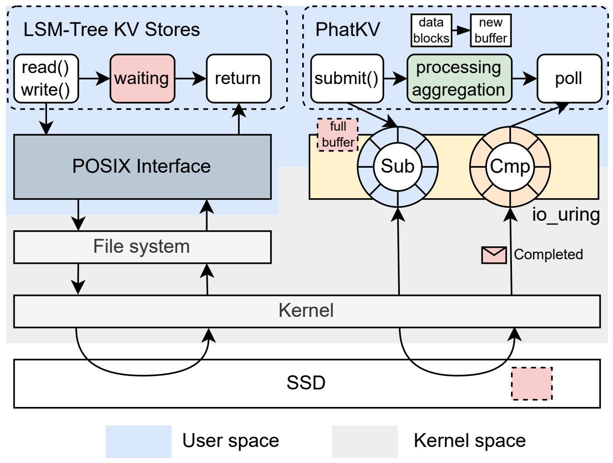

Fig. 8. **Comparison of IO framework.** Traditional LSM-Tree KV stores leverage POSIX-style synchronous IOs and suffer from stall for waiting the completion of IO. PhatKV uses the new efficient io_uring and can continue processing when the kernel is processing the IO requests.

multiple updates is guaranteed by upper layer transactions, such as the TransactionDB of RocksDB.

To minimize disk IO, log writes are done in batch whenever possible. For instance, when inserting a new key that requires creating a new f-hash entry, both the insertion of the KV item and the addition operation of the f-hash are recorded together in the consistency log. Similarly, modifications of all f-hash structures during the aggregation process are also persistently recorded in batches. Invalid data occurs when the newer modifications overlap the older ones. To eliminate invalid data in the consistency log, we do log refresh to invalidate the current log and create a new one by persisting all in-memory structures.

D. Asynchronous Aggregation Framework

The advent of modern SSDs with high throughput and low latency has shifted the bottleneck of KV stores from sluggish IO to the overhead of the software stack. Existing LSM-Tree KV stores leverage POSIX-style synchronous IO interface and incur two types of overheads: 1) redundant data copy: the data is copied between user space and kernel space; 2) synchronous overhead: working threads must wait for the completion of IO requests, which consumes CPU time. To take full advantage of the capabilities of modern SSDs, we leverage the asynchronous io_uring [10] interface to access data on the SSD. The io_uring is a standard Linux asynchronous IO interface. It leverages the work queue and completion queue, a pair of circular buffers that are shared by user-space applications and the Linux kernel, to asynchronously process IO requests. The io_uring interface has higher performance than the Linux AIO interface, and is more program-friendly than the SPDK library.

Based on the io_uring interface, we design the asynchronous framework for the aggregation, as shown in Figure 8. Specifically, a 4 KB buffer is used for storing KV entries generated by the aggregation. Each time the buffer is filled, it will be submitted to the work queue of io_uring during the aggregation process. After that, we immediately resume the aggregation without having to wait for the IO requests to complete. Finally, upon the completion of processing, the aggregation thread will check the pending IO requests and make sure that all pending IOs have been finished. Therefore, SSD IO and calculation overlap in the aggregation of PhatKV, enhancing the efficiency of aggregation.

IV. Discussion

Usability for other storage devices. PhatKV is designed for fast block storage devices. The piece hash index balances the large memory footprint of storing all the KV index entries with an in-memory hash table and the low efficiency of storing all the KV index entries with an on-disk hash table. The three-stage aggregation strategy reduces the overhead of arranging KV data on disk, where fine-grained data modifications are expensive. The asynchronous IO framework further enhances the utilization of fast SSD. For HDD, the piece hash index and three-stage aggregation are still effective, while the asynchronous IO is meaningless. For NVM, the byte-addressability makes it easy to achieve fine-grained data access. Thus, it is better to explore the performance of NVM by designing new index structures and data management strategies.

File system metadata operations with PhatKV. Metadata operations are transferred into KV operations of PhatKV by the file system. Locating a file or directory is conducted by recursively parsing each directory along the path via Get of PhatKV. Creating a file or directory will first locate the parent directory through the process of query. Then, a new entry with the new file is inserted. If the pinode is not found in the p-hash, a new f-hash will be constructed. If a file is deleted, the corresponding KV entry is also deleted. In PhatKV, deleting a key leads to inserting the same key with a tombstone as the value. It will be finally removed from PhatKV by the deep aggregation. Directory read is processed by fetching all the data blocks of a pinode via the in-memory data block set. After that, KV entries in the data blocks are parsed and redundant entries will be checked to find out invalid entries.

Recovery. PhatKV recovers from a system crash by reconstructing the in-memory data structures via the consistency log. PhatKV replays all the modifications recorded in the consistency log. The more modifications in the consistency log lead to longer time for rebuilding the in-memory data structures. Thus, the time for recovery is decided by two factors: (1) the interval of log refresh; and (2) the intensity of update operations.

V. Evaluation

In this section, we conduct experiments to evaluate the performance of PhatKV and compare it with state-of-the-art KV stores. We evaluate the fundamental KV operation performance by the micro-benchmark, and also analyze the performance in more depth by collecting detailed metrics in section V-C. Then, we present the results of widely used YCSB benchmarks in section V-D. Next, we integrate these KV stores into a file system and show how PhatKV improves the efficiency of file system metadata operations in section V-E. Finally, we discuss on the impact of f-cache size, memory overhead, and improvement of the IO interface in section V-F.

A. Environment

The evaluations are carried out on a server using two Intel Xeon Gold 5218(R) processors. The server has 64 GB DRAM and 480 GB Intel Optane 900P series SSD (Random Read: 540 KOPS, Random Write: 121 KOPS, Sequential Read: 2240 MB/s, Sequential Write: 1910 MB/s). The server operates on Ubuntu version 20.04, running kernel version 5.15.

B. Comparison and setup

We compare PhatKV with two LSM-Tree KV stores, LevelDB and RocksDB, and an in-place-update KV store, i.e., KVell.

LevelDB and RocksDB. The MemTable size and SSTable size are both configured to be 64 MB, with a block cache capacity of 1 GB. The bloom filter is enabled with 10 bits/key. All the SSTable files are allowed to be opened, and thus all the index blocks of SSTables are cached in memory. Although this configuration increases memory usage, it enhances search performance. Additionally, we optimize RocksDB by setting the maximum number of MemTable to 4 and the number of background threads to 8.

KVell. KVell is a KV store designed for modern SSD. It stores KV entries on SSD without sorting to eliminate the compaction overhead of LSM-Tree and manages all the KV entries with an in-memory B+Tree. KVell processes requests asynchronously via the pending request list. Foreground threads put KV requests into the request lists and wait for the work threads to process them. We configure KVell using the default settings provided by their open-sourced code, including an IO queue depth of 16 and a pending requests list size of 256. Moreover, we allocate a page cache size of 1 GB, matching the block cache size used by LevelDB and RocksDB.

PhatKV. We configure PhatKV according to RocksDB. The f-cache size is configured to be 1 GB, and the number

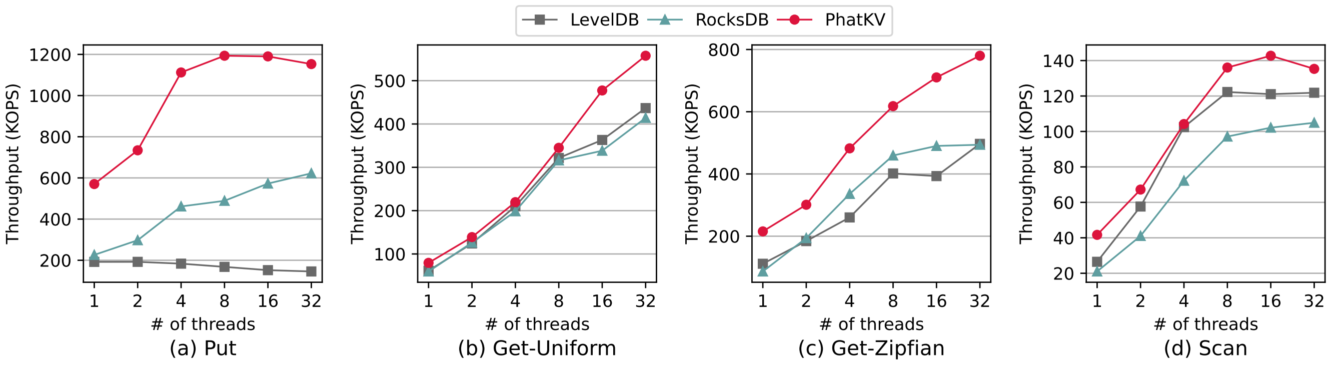

Fig. 9. **Throughput of micro-benchmarks.** Figure(a)-(d) show the throughput of LevelDB, RocksDB, and PhatKV under varied numbers of threads.

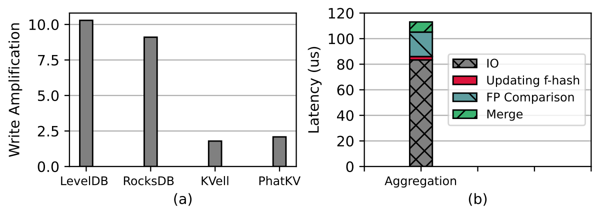

Fig. 10. **Detailed analysis of put operations.** Figure(a) shows the write amplification of loading 160M KV entries. Figure(b) shows the latency breakdown of PhatKV's background operations.

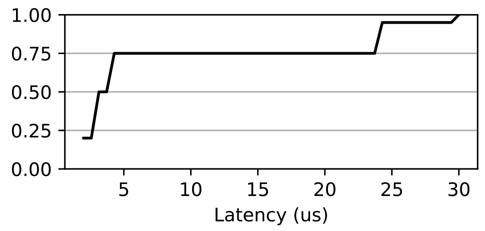

Fig. 11. **CDF of the get operations' latency.** This figure shows the latency of accessing KV entries with the same pinode in PhatKV.

of aggregation threads is set to 4, which provides sufficient resources for achieving optimal performance. We also implement PhatKV-async, where KV requests are put into a request queue and asynchronously processed by other work threads. Compared to synchronously processing KV requests via foreground threads (LevelDB and RocksDB), the asynchronous implementation achieves stable throughput with a varied number of work threads. So, we compare KVell and PhatKV separately for fairness.

C. Micro-benchmark

We first evaluate the throughput of three representative types of KV operations, i.e., Put, Get, and Scan, using a micro-benchmark. First, we insert 160 million KV entries into each KV store, followed by conducting either 20 million get operations or 1 million scan operations. The workload's keys consist of an 8-byte pinode and an 8-byte hash value, while the value size is set to 128 bytes, equivalent to the size of the file stat. Furthermore, the number of keys sharing the same pinode is set to 20, approximately equal to the number of files typically in a directory.

1) Put: Figure 9(a) illustrates the put throughput with varying numbers of threads. PhatKV attains the highest throughput under 8 threads, surpassing RocksDB and LevelDB by a factor of 2.5× and 7.9×, respectively. Due to its three-stage aggregation, PhatKV reduces write amplification. Additionally, PhatKV achieves better performance when utilizing only 4 background aggregation threads compared to RocksDB. This highlights how the aggregation in PhatKV contributes to reducing CPU resource consumption. Furthermore, the usage of io_uring for asynchronous IO enhances the efficiency of aggregations, avoiding delays of the foreground put operations due to untimely aggregation buffer flushes. Because LevelDB only has one thread for background compaction, which becomes the bottleneck for foreground operations, it has the lowest put throughput.

Analysis of aggregation. To analyze the effect of the aggregation on reducing write amplification, we record the write amplification of each KV store while loading 160 million KV entries. As shown in Figure 10(a), the write amplifications for LevelDB and RocksDB are 10.3 and 9.1, respectively. On the contrary, PhatKV has a write amplification of only 2.1, which is 0.2× lower than LevelDB and RocksDB. The write amplification comes from two aspects: 1) writing data blocks by the aggregation processes; and 2) persisting the updated f-hash structures after an aggregation operation. We further break down the time spent by the aggregation threads of PhatKV as shown in Figure 10(b). The calculation time for PhatKV is 0.18× lower than LevelDB because PhatKV does not need to merge and sort all the KV entries as the compaction does. Instead, PhatKV only merges a small portion of KV entries in the deep aggregation stage by comparing the fingerprints of KV entries in the data blocks. Thus, the FP comparison accounts for 64.4% of the total time for all the calculations of PhatKV's aggregation.

2) Get: We analyze the performance of the get operation under two workload distributions: uniform workload and

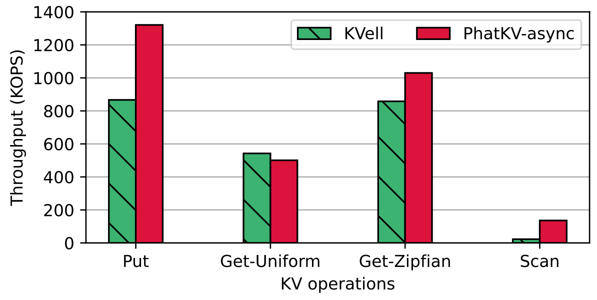

Fig. 12. **Micro-benchmark results of KVell.** This figure shows the throughput comparison between KVell and PhatKV-async.

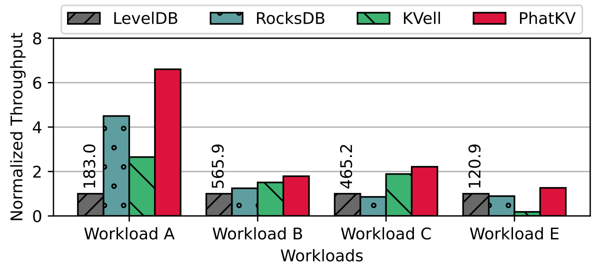

Fig. 13. **YCSB.** This figure shows the throughput of KV stores normalized to LevelDB. Four workloads with different read/write ratios are included: 1) Workload A: 50% read and 50% write; 2) Workload B: 95% read and 5% write; 3) Workload C: read-only workload; 4) Workload E: 95% scan and 5% write. All the workloads are under Zipfian distribution. The number on the bars of LevelDB is its real throughput.

skewed workload (zipfian). In the case of the uniform distribution, as shown in Figure 9(b), PhatKV only outperforms LevelDB by 30%. The overhead of loading and evicting f-hash structures from and to SSD is what we believe to be the cause of the slight improvement, as it lessens the benefit of the piece hash index. However, since the micro-benchmark generates requests randomly without any locality, this scenario can be considered as the worst-case for PhatKV. On the other hand, under the zipfian distribution, as shown in Figure 9(c), PhatKV outperforms LevelDB and RocksDB by 2.0× and 2.5×, respectively. In the skewed distribution, key lookup through f-hash is more efficient than using the index block of the LSM-Tree because of the increased hit ratio of the f-cache. The advantage of PhatKV becomes more pronounced when the LSM-Tree is unable to cache all its index blocks in memory. RocksDB performs worse than LevelDB since the increased number of memtables also increases the latency of searching a key.

Access latency under a directory. We further examine the latency of searching KV entries from the same directories to show how the piece hash index accelerates file access. As shown in Figure 11, 75% of the get requests can be served in 2 μs to 4 μs by leveraging the piece hash index and f-cache. For the other requests, the increased latency comes from 1) loading f-hash structures from SSD because of the miss of f-cache; and 2) loading data blocks from SSD because of the miss of data block cache.

3) Scan: PhatKV achieves 1.6× and 2.0× higher scan throughput than LevelDB and RocksDB, as shown in Figure 9(d). The reason for the improvement is that PhatKV can co-locate KV entries of a directory thanks to aggregation, which enables it to retrieve all the KV entries by directly reading all the target SSD pages. In comparison, LSM-Tree is less efficient than PhatKV because it needs to merge and sort the results of the data blocks from multiple SSTables. Moreover, PhatKV can achieve greater improvement than LSM-Tree as the directory size increases.

4) Comparison with KVell: Figure 12 compares the throughput of PhatKV and KVell for all the three types of KV operations. PhatKV's put throughput is 1.5× higher than KVell's because PhatKV incurs fewer random writes. As depicted in Figure 10(a), KVell has 0.86× lower write amplification than PhatKV because KVell doesn't move KV entries on SSD. This eliminates additional SSD IOs but incurs overhead for future scan operations. PhatKV has a 0.9× lower throughput compared to KVell under a uniform get workload because KVell stores the entire B+Tree in memory, whereas PhatKV's swapping in and out the f-hash causes a certain number of SSD IOs. However, the total memory occupied by KVell's B+Tree after loading 160 million entries is approximately 4.3 GB, which is four times larger than the f-cache. PhatKV achieves a 1.2× higher throughput than KVell under skewed workloads because of the superior efficiency of the piece hash index over the B+Tree. Finally, KVell shows the lowest scan performance, which is 0.2× lower than PhatKV, among the four KV stores. We attribute this to the unordered placement of KV entries in KVell. This unordered placement causes KV entries with the same pinode to scatter across slab files on SSD. Although B+Tree's scan operations are efficient, each retrieval of the KV data according to the address stored in B+Tree requires a random read on SSD. As a result, the efficiency of KVell is reduced by excessive random reads.

D. YCSB

YCSB [16] is a widely used macro-benchmark suite delivered by Yahoo!. We evaluate the performance of all KV stores under the YCSB workloads with the same initialization progress as described in Section V-C. We consider the three workloads with different read/write ratios (i.e., A, B, C) with the limitation of the KVell and the workloads for scan (i.e., E). We use 16 threads in this part to present the best performance. We show the throughputs normalized to LevelDB in Figure 13, and the results are consistent with those in the micro-benchmark. Because of the efficient piece hash index and light-weight aggregation strategy, PhatKV performs best in all workloads and outperforms LevelDB 6.6×, 1.8×, 2.2×, and 1.3× in workload A, B, C, and E, respectively. In workload A, RocksDB outperforms the KVell 1.7× because the new updates are handled by the Memtable while KVell must update the B+Tree structure. KVell achieves the worst performance in Workload E because of its inefficient scan operation.

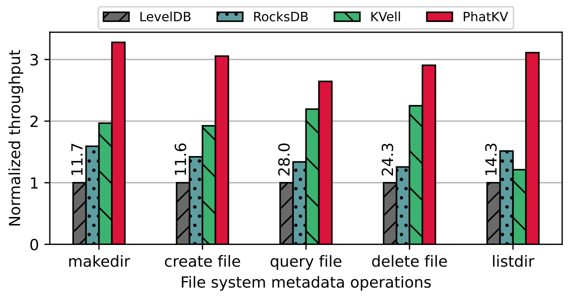

Fig. 14. **Normalized throughput of file system metadata operations.** This figure shows the throughput of file system metadata operations normalized to Tablefs-LevelDB. The numbers on the bars of LevelDB are the throughputs of Tablefs-LevelDB in KOPS.

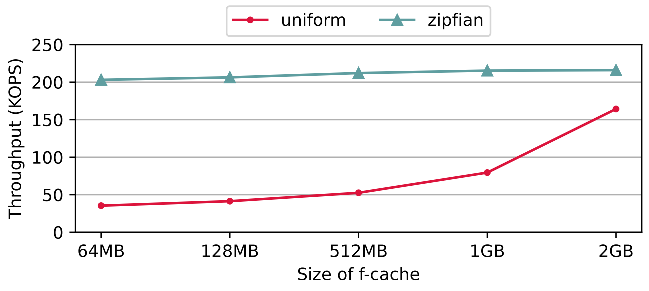

Fig. 15. **Impact of the f-cache size.** This figure shows the Get throughput of PhatKV under different f-cache size.

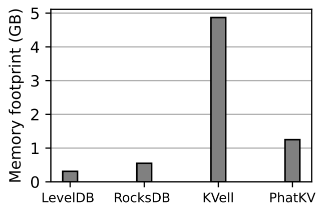

Fig. 16. **Memory overhead.** This figure shows the memory footprint of the four KV stores.

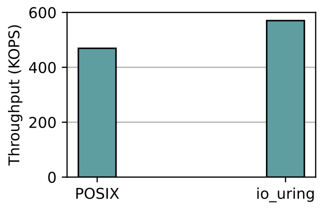

Fig. 17. **Effectiveness of io interface.** This figure demonstrates the write throughput of PhatKV under POSIX interface and io_uring interface.

E. File system metadata operations

We further compare the efficiency of typical metadata operations, such as file creation, file querying, file deletion, and listing directories, on the file system using different KV stores as the metadata storage engines. We select Tablefs [2], an open-source file system built on top of KV store, as our testbed. We construct two types of workloads for different metadata operations: 1) directory workload for the mkdir and listdir operations, consisting of a total of 50 million directories with a maximum depth of 19; 2) file workload for file creation, file querying, and file deletion, comprising 10,000 directories and 50 million files with a maximum depth of 12. The results are presented in Figure 14.

PhatKV outperforms LevelDB, RocksDB, and KVell on the mkdir and file creation operations with 3.1×, 2.1×, and 1.6× higher throughput, respectively. The mkdir and file creation operations involve the recursive parsing of directory paths, which involves both get and put operations. PhatKV performs faster put operations due to the three-stage aggregation, and also achieves faster queries through the piece hash index. Moreover, PhatKV maintains a high cache hit ratio due to the directory locality of metadata workloads. PhatKV outperforms LevelDB and RocksDB on the file query operations with 2.6× and 2.0× higher throughput, respectively. Since file query operations only involve the get operation, PhatKV achieves higher performance with a more efficient piece hash index. KVell performs better than LevelDB and RocksDB because searching in a B+Tree is more efficient than the index block of LSM-Tree. However, PhatKV outperforms KVell because the hash index of PhatKV is even more efficient than B+Tree searching. The file deletion operation is processed by first locating the target files and then deleting the KV entries. This is similar to the file creation operation because deleting a KV entry involves putting a tombstone marker. The tombstone marker and obsolete KV entries are removed by the compaction or aggregation threads later. Finally, PhatKV achieves 3.1×, 2.1×, and 2.6× higher throughput on the listdir operation. The efficiency of listdir operations is determined by both get operation and scan operation. The aggregation enables PhatKV to achieve a more efficient scan by grouping the KV entries of a directory together.

F. Detail Analysis

In this section, we first discuss the impact of the f-cache size on the performance of get operations. Then, we compare the memory overhead of the mentioned four KV stores. Finally, we present the improvement of PhatKV by using the new IO interface.

The size of f-cache. The f-cache of PhatKV manages the in-memory f-hash structures and evicts f-hash structures to SSD to limit the memory footprint when necessary. To investigate how the size of f-cache affects the performance of PhatKV's get operations, we run a micro-benchmark with different f-cache sizes varying from 64 MB to 2GB. Figure 15 shows the results. The size of f-cache has a significant impact on the get operation performance under a uniform workload because a larger f-cache can keep more f-hash structures in memory, reducing SSD IOs for evicting and loading f-hash. On the contrary, under the skewed workload, the get operation's performance remains stable against the variation of f-cache sizes because the smallest f-cache is sufficient to keep all the accessed f-hash structures in memory. Additionally, optimizing the efficiency of the cache management algorithm can raise the f-cache's hit ratio. This can reduce SSD IOs for transferring f-hash between SSD and memory, and improve the efficiency of get operation. We leave the issues of optimizing the cache algorithm according to the file system workloads as our future work.

Memory overhead. The two-level piece hash index accelerates the KV searching by leveraging in-memory hash index.

Figure 16 shows the memory footprint of the four KV stores after loading 160 million KV entries. KVell consumes the most memory resources because of its adoption of in-memory B+Tree. The memory used by KVell is 3.9× larger than that in PhatKV. PhatKV consumes 2.3× and 4.2× more memory than LevelDB and RocksDB. PhatKV can easily adjust the cache size according to the workload characteristics. However, LSM-Tree KV stores may have worse search efficiency if they cannot keep all the SSTable metadata in memory.

IO interface. PhatKV improves SSD write efficiency by leveraging the asynchronous io_uring interface. We run the micro-benchmark to evaluate PhatKV by loading 160 million entries under the traditional POSIX interface and the io_uring interface. As demonstrated in Figure 17, PhatKV with the io_uring interface improves 1.2× put throughput. This improvement comes from 1) more efficient handling of IO requests by io_uring; 2) overlapping IO with processing by asynchronous interface.

VI. Related Works

1) File systems with KV store: KV stores are selected as the storage engine of file system metadata when building new fast and scalable file systems. Compared to traditional file systems, KV stores are deployed for the local file system to achieve more efficient IOs for small metadata. Tablefs [2] provides efficient metadata operations and small file access by leveraging LevelDB to consolidate the random IO caused by metadata updates and small files. KVFS [17] optimizes LSM-Tree by reducing write amplification via VT-Tree and builds the new KVFS based on it. betrfs [11] leverages TokuDB in a different way where the full paths of files are used as the keys. KEVIN [6] incorporates the new KVSSD and LSM-Tree to manage the file system metadata. For the distributed file systems, KV stores are typically deployed as an independent service such as metadata servers. IndexFS [3] is the distributed version of Tablefs and also takes advantage of LevelDB to manage its metadata in each metadata server. LocoFS [4] manages the metadata by LocoMeta [18] that is built based on Kyoto Cabinet. Techtonic [5] builds its metadata layer on the distributed ZippyDB, whose local storage is RocksDB. InfiniFS [1] stores its metadata on the underlying RocksDB. JuiceFS [19] is an open-sourced distributed file system that supports several metadata backends including Redis and TiKV. The metadata performance is however limited due to the fact that existing file systems use KV stores designed for general use cases and consider the file system level rather than the KV store.

2) LSM-Tree KV stores: LSM-Tree based KV stores, such as LevelDB and RocksDB, are optimized for fast write processing and are deployed as the backbones of various applications. Various optimizations are proposed for LSM-Tree to attain better write or read performance.

Optimizing compaction strategy. PebblesDB [20] adopts a tiering-style compaction strategy and leverages the "guard" to guarantee the partially sorted levels. SpliniterDB [21] combines LSM-Tree and Bϵ-Tree to build the new SPTϵ-tree. EvenDB [22] leverages the spatial-locality of KV workloads. The LSM-Tree compaction is accelerated in literature [23] by offloading it to FPGA hardware. WipDB [24] uses an approximately sorted list via in-place updates. BlockDB [25] accelerates the compaction by adopting a block-style compaction strategy. Pacman [26] optimizes the compaction according to the characteristics of persistent memory.

KV separation. Another way of reducing write amplification is KV separation strategy [7], [27], [28]. Among these works, DiffKV [7] handles KV with different sizes independently and proposes a compaction-triggered merge operation for efficiently managing the separated KV data.

Hybrid storage. With the emerging high-speed storage devices such as persistent memory or NVMe SSD, there are also considerable works aiming to optimize LSM-Tree with heterogeneous storage hierarchy. [29] accelerates RocksDB by using PM cache. SpanDB [30], MatrixKV [31], and PM-Blade [32] transfer the higher levels of LSM-Tree into faster devices to accelerate data writing. [33] stores the KV data on NVM with high access frequency.

Read optimization. Several works aim to accelerate key searching in LSM-Tree. AC-Key [9] proposes a new cache management algorithm. Bourbon [34] integrates a learned index to accelerate locating a key on SSD. Chunky [35] proposes a new global cuckoo filter to map each data entry to an auxiliary address corresponding to its location. REMIX [36] accelerates the LSM-Tee Scan by building a globally sorted view of KV data spanning multiple table files.

3) In-place Update KV stores: Modern SSDs, such as NVMe or Optane SSD, have shifted the bottleneck in system performance from SSD to the software overhead, which has prompted the development of in-place-update KV stores. LeanStore [37] leverages the swizzling pointer and low-overhead replacement strategy to efficiently manage KV data on SSD. KVell [8] eliminates sorting on SSD and indexes all the KV data on SSD by an in-memory B+Trees. Treeline [38] accelerates the efficiency of KV access by key cache. It reduces the memory overhead of in-memory B+Tree by using a learned index and boosts the scan performance by grouping KV data on SSD.

Current optimizations focus more on general-purpose KV stores rather than file system metadata. PhatKV can achieve a higher efficiency if it is designed with the characteristics of file system metadata workloads in consideration. Moreover, some optimizations are orthogonal to PhatKV, including hierarchical storage, cache algorithms, and index structures. PhatKV may achieve a better performance with these technologies.

VII. Conclusion

In this paper, we reveal the KV workload characteristics of the file system metadata operations and the limitations of the widely used LSM-Tree when used as the storage engine for file system metadata on SSD. We propose PhatKV, which aims to work as the metadata engine of KV-based file systems. PhatKV leverages a piece hash index and a three-stage aggregation to enable efficient file system metadata operations. Additionally, PhatKV also leverages the new asynchronous io_uring interface to realize more efficient SSD IOs. The results demonstrate that PhatKV outperforms LevelDB and RocksDB by 1.2× and 7.9×, respectively. It also achieves 1.2× and 1.5× higher throughput compared to KVell, and a 4× less memory footprint compared to KVell. Moreover, the file system using PhatKV achieves 1.6× and 3.1× improvements on metadata operations than that using LevelDB and RocksDB.

Acknowledgment

This work was sponsored by the National Natural Science Foundation of China under Grant No.62072196, the National Key Research and Development Program of China under Grant No.2023YFB4502701, the Key Research and Development Program of Guangdong Province under grant No. 2021B0101400003, the Creative Research Group Project of NSFC No.61821003.

References

[1] W. Lv, Y. Lu, Y. Zhang, P. Duan, and J. Shu, "{InfiniFS}: An efficient metadata service for {Large-Scale} distributed filesystems," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pp. 313–328, 2022.

[2] K. Ren and G. Gibson, "Tablefs: Embedding a nosql database inside the local file system," in *2012 Digest APMRC*, pp. 1–6, IEEE, 2012.

[3] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 237–248, IEEE, 2014.

[4] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "Locofs: A loosely-coupled metadata service for distributed file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[5] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, *et al.*, "Facebook's tectonic filesystem: Efficiency from exascale," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pp. 217–231, 2021.

[6] J. Koo, J. Im, J. Song, J. Park, E. Lee, B. S. Kim, and S. Lee, "Modernizing file system through in-storage indexing," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 75–92, 2021.

[7] Y. Li, Z. Liu, P. P. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui, "Differentiated {Key-Value} storage management for balanced {I/O} performance," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 673–687, 2021.

[8] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 447–461, 2019.

[9] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du, "{AC-Key}: Adaptive caching for {LSM-based}{Key-Value} stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 603–615, 2020.

[10] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi, "Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, pp. 120–127, 2022.

[11] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, *et al.*, "Betrfs: A right-optimized write-optimized file system," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pp. 301–315, 2015.

[12] Y. Zhang, J. Zhou, X. Min, S. Ge, J. Wan, T. Yao, and D. Wang, "Petakv: Building efficient key-value store for file system metadata on persistent memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 843–855, 2022.

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[14] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 209–223, 2020.

[15] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 631–644, 2015.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[17] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pp. 17–30, 2013.

[18] S. Li, F. Liu, J. Shu, Y. Lu, T. Li, and Y. Hu, "A flattened metadata service for distributed file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2641–2657, 2018.

[19] "Juicefs." https://github.com/juicedata/juicefs.

[20] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 497–514, 2017.

[21] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson, "{SplinterDB}: Closing the bandwidth gap for {NVMe}{Key-Value} stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 49–63, 2020.

[22] E. Gilad, E. Bortnikov, A. Braginsky, Y. Gottesman, E. Hillel, I. Keidar, N. Moscovici, and R. Shahout, "Evendb: optimizing key-value storage for spatial locality," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[23] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "Fpga-based compaction engine for accelerating lsm-tree key-value stores," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1261–1272, IEEE, 2020.

[24] X. Zhao, S. Jiang, and X. Wu, "Wipdb: A write-in-place key-value store that mimics bucket sort," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 1404–1415, IEEE, 2021.

[25] X. Wang, P. Jin, B. Hua, H. Long, and W. Huang, "Reducing write amplification of lsm-tree with block-grained compaction," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 3119–3131, IEEE, 2022.

[26] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for {Log-Structured}{Key-Value} store on persistent memory," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 773–788, 2022.

[27] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[28] H. H. Chan, C.-J. M. Liang, Y. Li, W. He, P. P. Lee, L. Zhu, Y. Dong, Y. Xu, Y. Xu, J. Jiang, *et al.*, "{HashKV}: Enabling efficient updates in {KV} storage via hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 1007–1019, 2018.

[29] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "Improving performance of flash based {Key-Value} stores using storage class memory as a volatile memory extension," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 821–837, 2021.

[30] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "{SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 17–32, 2021.

[31] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "{MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} based {KV} stores with matrix container in {NVM}," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 17–31, 2020.

[32] Y. Zhang, H. Hu, X. Zhou, E. Xie, H. Ren, and L. Jin, "Pm-blade: A persistent memory augmented lsm-tree storage for database," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3363–3375, IEEE, 2023.

[33] L. Chen, R. Chen, C. Yang, Y. Han, R. Zhang, X. Zhou, P. Jin, and W. Qian, "Workload-aware log-structured merge key-value store for nvm-ssd hybrid storage," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 2207–2219, IEEE, 2023.

[34] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “From {WiscKey} to bourbon: A learned index for {Log-Structured} merge trees,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 155–171, 2020.

[35] N. Dayan and M. Twitto, “Chucky: A succinct cuckoo filter for lsm-tree,” in *Proceedings of the 2021 International Conference on Management of Data*, pp. 365–378, 2021.

[36] W. Zhong, C. Chen, X. Wu, and S. Jiang, “{REMIX}: Efficient range query for {LSM-trees},” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 51–64, 2021.

[37] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, “Leanstore: In-memory data management beyond main memory,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 185–196, IEEE, 2018.

[38] G. X. Yu, M. Markakis, A. Kipf, P.-Å. Larson, U. F. Minhas, and T. Kraska, “Treeline: an update-in-place key-value store for modern storage,” *Proceedings of the VLDB Endowment*, vol. 16, no. 1, pp. 99–112, 2022.